Data Analysis and Visualization in R (IN2339)

Exercise Session 1 - Solutions

Daniela Klaproth-Andrade, Julien Gagneur

Section 01 - R very basics

1. What is the sum of the first 100 positive integers? The formula for the sum of integers from 1 to n is $n(n+1)/2$. Define $n = 100$ and then use R to compute the sum from 1 to 100 using the formula. What is the sum?

```
n <- 100
sum_n <- n*(n+1)/2
sum_n
```

```
## [1] 5050
```

2. Now use the same formula to compute the sum of the integers from 1 to 1,000.

```
n <- 1000
sum_n <- n*(n+1)/2
sum_n
```

```
## [1] 500500
```

3. Look at the result of typing the following code into R:

```
n <- 1000
x <- seq(1, n)
sum(x)
```

```
## [1] 500500
```

Based on the result, what do you think the functions `seq` and `sum` do? You can use `help`.

a. `sum` creates a list of numbers and `seq` adds them up.
b. `seq` creates a list of numbers and `sum` adds them up.
c. `seq` creates a random list and `sum` computes the sum of 1 through 1,000.
d. `sum` always returns the same number.

```
# The correct answer is b
```

4. In math and programming, we say that we evaluate a function when we replace the argument with a given value. So if we type `sqrt(4)`, we evaluate the `sqrt` function. In R, you can evaluate a function inside another function. The evaluations happen from the inside out. Use one line of code to compute the logarithm, in base 10, of the square root of 100.

```
log10(sqrt(100)) ## or log(sqrt(100), base=10)
```

```
## [1] 1
```

5. Which of the following will always return the numeric value stored in `x`? You can try out examples and use the help system if you want.

a. `log(10^x)`
b. `log10(x^10)`
c. `log(exp(x))`
d. `exp(log(x, base = 2))`

```r
## The correct answer is c since log() computes by default natural logarithms. For example:
x <- 13
log(exp(x))
```

```
## [1] 13
```

6. What is the outcome of the following sum $1 + 1/2^2 + 1/3^2 + \cdots + 1/100^2$? Thanks to Euler, we know it should be close to $\pi^2/6$. Compute the sum and check that it is close to the approximation. Note that R has a variable for π stored as `pi`.

```r
sum(1/seq(1:100)^2)
```

```
## [1] 1.634984
```

```r
pi^2/6
```

```
## [1] 1.644934
```

Section 02 - Basic data

1. Load the US murders dataset from the `dslabs` package executing the following commands:

```r
library(dslabs)
data(murders)
```

Use the function `str` to examine the structure of the `murders` object. Which of the following best describes the variables represented in this data frame?

a. The 50 states and DC.
b. The murder rates for all 50 states and DC.
c. The state name, the abbreviation of the state name, the state's region, and the state's population and total number of murders for 2010.
d. `str` shows no relevant information.

```r
# The correct answer is c
str(murders)
```

```
## 'data.frame':    51 obs. of  5 variables:
##  $ state     : chr  "Alabama" "Alaska" "Arizona" "Arkansas" ...
##  $ abb       : chr  "AL" "AK" "AZ" "AR" ...
##  $ region    : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...
##  $ population: num  4779736 710231 6392017 2915918 37253956 ...
##  $ total     : num  135 19 232 93 1257 ...
```

2. What are the column names used by the data frame for these five variables?

```r
# We can use the function names()
names(murders)
```

```
## [1] "state"      "abb"        "region"     "population" "total"
```

```r
#alternatively we can also use the function colnames()
colnames(murders)
```

```
## [1] "state"      "abb"        "region"     "population" "total"
```

3. Use the accessor `$` to extract the state abbreviations and assign them to the variable `a`. What is the class of this object?

```r
a <- murders$abb # Access the column of a dataframe with $
class(a)  # get the class, also possible to use str()
```

```
## [1] "character"
```

4. We saw that the `region` column stores a factor. You can corroborate this by typing:

```r
class(murders$region)
```

With one line of code, use the function `levels` and `length` to determine the number of regions defined by this dataset.

```r
length(levels(murders$region))
```

```
## [1] 4
```

5. The function `table` takes one or multiple vectors and returns the frequency of each element. For example:

```r
x <- c('DC', 'Alabama', 'Florida', 'Florida', 'DC', 'DC')
table(x)
```

```
## x
## Alabama      DC Florida
##       1       3       2
```

You can quickly see how many states are in each region by applying this function. Use this function in one line of code to create a table of states per region.

```r
table(murders$region)
```

```
##
##     Northeast         South North Central          West
##             9            17            12            13
```

Section 03 - Working with vectors

1. Use the function `c` to create a vector with the average high temperatures in January for Beijing, Lagos, Paris, Rio de Janeiro, San Juan, and Toronto, which are 35, 88, 42, 84, 81, and 30 degrees Fahrenheit. Call the object `temp`.

```r
temp <- c(35, 88, 42, 84, 81, 30)
temp
```

```
## [1] 35 88 42 84 81 30
```

2. Now create a vector with the city names and call the object `city`.

```r
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro", "San Juan", "Toronto")
city
```

```
## [1] "Beijing"        "Lagos"          "Paris"          "Rio de Janeiro"
## [5] "San Juan"       "Toronto"
```

3. Use the `names` function and the objects defined in the previous exercises to associate the temperature data with its corresponding city.

```r
names(temp) <- city
temp
```

```
##        Beijing          Lagos          Paris Rio de Janeiro       San Juan
##             35             88             42             84             81
##        Toronto
##             30
```

4. Use the [and : operators to access the temperature of the first three cities on the list.

```
temp[1:3]
```

```
## Beijing   Lagos   Paris
##      35      88      42
```

5. Use the [operator to access the temperature of Paris and San Juan.

```
temp[c("Paris", "San Juan")]
```

```
##    Paris San Juan
##       42       81
```

6. Use the : operator to create a sequence of numbers $12, 13, 14, \ldots, 73$.

```
12:73
```

```
##  [1] 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
## [24] 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
## [47] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
```

7. Create a vector containing all the positive odd numbers smaller than 100.

```
seq(1, 99, 2)
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45
## [24] 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91
## [47] 93 95 97 99
```

8. Create a vector of numbers that starts at 6, does not pass 55, and adds numbers in increments of 4/7: 6, 6 + 4/7, 6 + 8/7, and so on. How many numbers does the vector have? Hint: use `seq` and `length`.

```
v <- seq(6, 55, 4/7)
length(v)
```

```
## [1] 86
```

9. What is the class of the following object `a <- seq(1, 10, 0.5)`?

```
a <- seq(1, 10, 0.5)
class(a)
```

```
## [1] "numeric"
```

10. What is the class of the following object `a <- seq(1, 10)`?

```
a <- seq(1, 10)
class(a)
```

```
## [1] "integer"
```

11. The class of `class(a<-1)` is numeric, not integer. R defaults to numeric and to force an integer, you need to add the letter `L`. Confirm that the class of `1L` is integer.

```
class(1L)
```

```
## [1] "integer"
```

12. Define the following character vector:

```r
x <- c("1", "3", "5")
```

and coerce it to get integers.

```r
x <- c("1", "3", "5")
x_int <- as.integer(x)
x_int
```

```
## [1] 1 3 5
```

Section 04 - sorting and ranking

For these exercises we will use the US murders dataset. Make sure you load it prior to starting.

```r
library(dslabs)
data("murders")
```

1. Use the `$` operator to access the population size data and store it as the object `pop`. Then use the `sort` function to redefine `pop` so that it is sorted. Finally, use the `[` operator to report the smallest population size.

```r
pop <- murders$population
pop <- sort(pop) # sorts ascending by default
pop[1] # smallest value at the first position
```

```
## [1] 563626
```

2. Now instead of the smallest population size, find the index of the entry with the smallest population size. Hint: use `order` instead of `sort`.

```r
pop <- murders$population
smallest_idx <- order(pop)
smallest_idx[1] #  index with the smallest population
```

```
## [1] 51
```

```r
pop[smallest_idx[1]] #  smallest population value
```

```
## [1] 563626
```

3. We can actually perform the same operation as in the previous exercise using the function `which.min`. Write one line of code that does this.

```r
which.min(murders$population)
```

```
## [1] 51
```

4. Now we know the population of the smallest state is and the row which represents it. What is the name of this state? Define a variable `states` to be the state names from the `murders` data frame. Report the name of the state with the smallest population.

```r
murders$state[which.min(murders$population)]
```

```
## [1] "Wyoming"
```

5. You can create a data frame using the `data.frame` function. Here is a quick example:

```r
temp <- c(35, 88, 42, 84, 81, 30)
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro",
          "San Juan", "Toronto")
city_temps <- data.frame(name = city, temperature = temp)
```

Use the `rank` function to determine the population rank of each state from smallest population size to biggest. Save these ranks in an object called `ranks`, then create a data frame with the state name and its rank. Call the data frame `my_df`.

```
ranks <- rank(murders$population)
my_df <- data.frame(state=murders$state, rank=ranks)
head(my_df)
```

```
##        state rank
## 1    Alabama   29
## 2     Alaska    5
## 3    Arizona   36
## 4   Arkansas   20
## 5 California   51
## 6   Colorado   30
```

Section 05 - Vector arithmetics

1. Have a look a the following lines of code:

```
temp <- c(35, 88, 42, 84, 81, 30)
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro",
          "San Juan", "Toronto")
names(temp) <- city
```

Use the `temp` vector to create a different vector `temp_F` that converts the temperature from Fahrenheit to Celsius. The conversion is $C = \frac{5}{9} \times (F - 32)$.

```
temp <- c(35, 88, 42, 84, 81, 30)
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro",
          "San Juan", "Toronto")
names(temp) <- city
temp_F <- 5/9 * (temp - 32)
temp_F
```

```
##        Beijing          Lagos          Paris Rio de Janeiro       San Juan
##       1.666667      31.111111       5.555556      28.888889      27.222222
##        Toronto
##      -1.111111
```

2. The `na_example` vector represents a series of counts. You can quickly examine the object using:

```
data("na_example")
str(na_example)
```

```
##  int [1:1000] 2 1 3 2 1 3 1 4 3 2 ...
```

However, when we compute the average with the function `mean`, we obtain an `NA`:

```
mean(na_example)
```

```
## [1] NA
```

The `is.na` function returns a logical vector that tells us which entries are `NA`. Assign this logical vector to an object called `ind` and determine how many `NA`s does `na_example` have.

```
ind <- is.na(na_example)

table(ind)
```

```
## ind
## FALSE  TRUE
##   855   145
```

```
# alternatively we can just sum the boolean
# sum(ind)
# or subset the vector to contain only the NA's and obtain its length
# length(ind[ind])
# to obtain the number of NA's.
```

3. Now compute the average again, but only for the entries that are not `NA`. Hint: remember the `!` operator.

```
mean(na_example[!ind])
```

```
## [1] 2.301754
```

```
# the same can be achieved, by simply setting the na.rm argument of the mean
# function to TRUE.

# mean(na_example, na.rm=TRUE)
```